John von Neumann Institute for Computing

OpenMP Implementation of the Householder Reduction for Large Complex Hermitian Eigenvalue Problems

Andreas Honecker, Josef Schüle

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 271-278, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

http://www.fz-juelich.de/nic-series/volume38

OpenMP Implementation of the Householder Reduction for Large Complex Hermitian Eigenvalue Problems

Andreas Honecker[1] **and Josef Schüle**[2]

[1] Institut für Theoretische Physik, Georg-August-Universität Göttingen
37077 Göttingen, Germany
E-mail: ahoneck@uni-goettingen.de

[2] Gauss-IT-Zentrum, Technische Universität Braunschweig, 38106 Braunschweig, Germany
E-mail: j.schuele@tu-bs.de

The computation of the complete spectrum of a complex Hermitian matrix typically proceeds through a Householder step. If only eigenvalues are needed, this Householder step needs almost the complete CPU time. Here we report our own parallel implementation of this Householder step using different variants of C and OpenMP. As far as we are aware, this is the only existing parallel implementation of the Householder reduction for complex Hermitian matrices which supports packed storage mode. As an additional feature we have implemented checkpoints which allow us to go to dimensions beyond 100 000. We perform runtime measurements and show firstly that even in serial mode the performance of our code is comparable to commercial libraries and that secondly we can obtain good parallel speedup.

1 Introduction

There are problems which require the complete diagonalization of big complex Hermitian matrices. For example, in theoretical solid-state physics, the computation of thermodynamic properties of certain quantum many-body problems boils down to the computation of the complete spectrum of eigenvalues of the Hermitian matrix representing the Hamiltonian[1–3] (eigenvectors are not needed for this specific purpose, but would be required for the computation of other properties). In this application, the original problem is split into smaller pieces using symmetries[4,5]. In particular, the aforementioned problems are lattice problems and we can use Fourier transformation to exploit translational symmetry. This yields a range of matrix eigensystems where the matrices of largest dimension turn out to be complex. Thus, the diagonalization of Hermitian matrices constitutes the bottleneck of such computations. Although the individual matrices are originally sparse, they are treated as dense matrices since all eigenvalues are needed, and the standard algorithms for solving this problem[6–8] proceed through a stage where the matrix becomes densely populated.

Libraries like LAPACK[9] provide diagonalization routines such as `zheev` for dense complex Hermitian matrices. Such routines typically perform first a Householder reduction to tridiagonal form and then use QR-/QL-iteration to solve the tridiagonal problem[6,10]. The applications mentioned above[1–3] do not need eigenvectors. Therefore, almost all CPU time and memory is spent in the Householder step. Furthermore, if eigenvectors are not needed, one can store the matrix in a so-called packed form which exploits symmetry of the matrix and stores only one triangular half. Available library routines work well in low dimension n, but memory requirements grow as n^2 and CPU time grows as n^3 for larger dimensions. More precisely, approximately $8\,n^2$ bytes are needed to store a complex double-precision matrix in packed form and the Householder reduction of such a matrix

requires approximately $16\,n^3/3$ floating-point operations. For example, in dimension $n = 40\,000$ we need 12 GByte to store the complex matrix in packed form, while full storage would already require 24 GByte main memory. Thus, for complex dimensions $n \gtrsim 40\,000$ substantial fractions of the main memory even of current high-performance computers are going to be used. Consequently, a parallelized code should be used to utilize the CPUs attached to this memory and reduce runtime.

There are different parallel diagonalization routines supporting distributed memory and several comparative studies of such libraries exist (see, e.g., Refs. 11–13). We will not attempt yet another comparative study, but just mention a few relevant features. ScaLAPACK[14] and variants like PESSL[15] do unfortunately not support packed matrices and even require two copies for the computation of eigenvectors. In addition, these libraries may have problems with numerical accuracy[11]. PeIGS supports packed matrices, but it does not provide diagonalization routines for complex matrices[16]; the necessary conversion of a complex Hermitian to a real symmetric eigenvalue problem wastes CPU-time and a factor 2 of main memory. There are further parallel matrix diagonalization routines (see, e.g., Refs. 17–20), but we do not know any parallel implementation of an eigensolver for complex Hermitian matrices which contends itself with the minimal memory requirements.

On shared memory parallel computers, the diagonalization routines contained e.g. in LAPACK may call a parallel implementation of the basic linear algebra package BLAS. However, this does not yield good parallel performance since parallelization should set in at a higher level. While inspection of the Householder algorithm[6–8,21] shows that it is possible to parallelize the middle of three nested loops, we are not aware of any such parallelization for a shared-memory parallel machine.

Here we report our own efforts at a parallel implementation of the Householder reduction in C using OpenMP[22]. In Section 2 we first discuss our parallelization strategy. Then we present performance measurements both of the serial and the parallel code in Section 3. Finally, Section 4 summarizes the current status and discusses future plans.

2 Implementation of the Householder Algorithm

We start from a serial implementation of the standard Householder algorithm[6–8] in C99. A complete version of the code is available from Ref. 23. This code has been inspired by the routine `tred2` of Ref. 21, but is has been generalized to complex numbers, thoroughly optimized for modern CPUs, and prepared for parallelization.

Let us discuss our parallelization strategy using a part of the code as an example. The fragment which performs a reduction of the Hermitian matrix `m` in C99 is shown in Listing 1.

First, we observe that dependencies inside the loop over `j` can be eliminated by performing the operations on `p` in a separate loop. It is then possible to parallelize the loop over `j`, i.e. the middle of three nested loops, with OpenMP by adding a `#pragma omp parallel for` in front of it. One problem with such a naïve approach is that the length of the innermost loop depends on `j` such that we run the risk that the work is not distributed equally across the different threads. It is possible to ensure load balancing by introducing an auxiliary variable which explicitly runs over the threads. However, a second problem remains. Namely, the innermost loop over `k` should be able to reuse the vector `p` and the row `i` of the matrix `m` from the Level 2 cache of the CPU. For dimensions `dim` around

Listing 1. Householder algorithm: Hermitian matrix reduction

```
for (i=dim-1; i>=1; i--) {
  :
    for (j=0; j<i; j++) {
      f = conj(m[i][j]);
      g = p[j] = p[j]-hh*f;
      cptr1 = p; cptr2 = m[j]; cptr3 = m[i];
      for (k=0; k<=j; k++)
        *cptr2++ -= (f * conj(*cptr1++) + g * *cptr3++);
    }
  :
}
```

33 000 the memory requirements for these data start to exceed 1 MByte such that they fail to fit into the Level 2 cache for larger dimensions, leading to a substantial degradation of the performance of such a code for big matrices.

Therefore, we proceed differently. Namely, we split the innermost loop over `k` into sufficiently small chunks and pull the loop over the chunks outside the loop over `j`. A parallel version of the above code fragment then looks as follows:

Listing 2. Parallel matrix reduction

```
for (i=dim-1; i>=1; i--) {
  :
    for (j=0; j<i; j++) {
      p[j] -= hh*conj(m[i][j]);
      nchunks = compute_chunks(sizeof(complex double), i-1);
#pragma omp parallel for private(chunk,j,k,f,g,cbegin,cend,
                    cendt,cptr1,cptr2,cptr3,pptr) if(nchunks>1)
      for (chunk=0; chunk<nchunks; chunk++) {
        cbegin = chunks[chunk].begin;
        cendt = chunks[chunk].end;
        pptr = p+cbegin;
        for (j=0; j<i; j++) {
          f = conj(m[i][j]);
          g = p[j];
          cptr1 = pptr;
          cptr2 = m[j]+cbegin;
          cptr3 = m[i]+cbegin;
          cend = (j<cendt)?j:cendt;
          for (k=cbegin; k<=cend; k++)
             *cptr2++ -= (f * conj(*cptr1++) + g * *cptr3++);
        }
      }
    }
  :
}
```

The administrative routine `compute_chunks` has the task of splitting the segment of size `i` into chunks such that two constraints are obeyed. Firstly, each chunk should fit into Level 2 CPU cache. For this reason the size of an individual element is passed as an argument. Secondly, the load is to be balanced knowing that the inner loop is restricted to $\mathtt{k} \leq \mathtt{j}$. There are two further variants of this routine: `compute_eq_chunks` performs the same task for the situation where the inner loop also runs to `i-1` and

	2.4 GHz Intel Core2		1.9 GHz IBM Power5		1.6 GHz Itanium2	
variant	gcc 4.1.1	icc 10.0	gcc 4.0.2	xlc 8.0	gcc 4.1.0	icc 9.1
C99	5.18	4.60	7.05	17.03	20.42	8.04*
C++	5.11	44.23	7.05	10.18	26.70	16.87
plain C	5.48	5.46	7.94	6.43*	28.74	14.68
SSE3	4.39*	4.41				

Table 1. Runtimes in seconds for the serial computation of eigenvalues for a "small" complex matrix in dimension $n = 1184$. Different rows are for different variants of our code, different columns for different CPUs and compilers (GNU gcc, Intel icc and IBM xlc). Note that runtimes include overhead, e.g., for initialization and diagonalization of the resulting tridiagonal matrix.

`compute_min_chunks` just computes chunks such that they fit into Level 2 CPU cache. With these administrative routines we can then follow the same strategy as above and parallelize a total of three loops which are needed for the reduction to tridiagonal form and two additional loops which compute the transformation matrix which is needed if eigenvectors are also desired.

The following additional features of our implementation may be worthwhile mentioning. Firstly, at the beginning of each outermost loop over `i` it is possible to checkpoint the computation by writing the updated part of the matrix `m` and some additional data onto hard disc. We have implemented such checkpoints with a second set of administrative routines. Secondly, the matrix `m` is implemented as a vector of pointers to its rows. On the one hand, this renders it unnecessary to store the complete matrix consecutively in memory and also allows convenient access to the matrix elements in a lower triangular packed storage mode. On the other hand, in combination with the checkpoint it becomes possible to return memory to the system for those parts of the matrix where the computation is completed, allowing part of the computation to run with reduced memory requirements.

3 Performance

We start with tests of the serial performance of our code. For this purpose we use a "small" complex Hermitian matrix of dimension $n = 1184$ which can be downloaded from Ref. 23. This matrix needs about 11 MByte if stored in packed form and should therefore be big enough *not* to fit completely into the Level 2 CPU cache.

First, we have compared the public domain Gnu C-compiler[24] with compilers provided by the vendors on different platforms as well as different variants of the implementation of the Householder algorithm, namely the C99 version discussed in Section 2, a C++ version using the `complex<double>` data type from the Standard Template Library, a version with complex numbers hand-coded in plain C, and a version with complex numbers hand-coded in inline-assembler for CPUs supporting SSE instructions. Results are shown in Table 1. One observes that variations of runtimes by a factor 3 on the same CPU are not uncommon depending on the version of our code and more noteworthy on the compiler. On the Intel Core2 we obtained the best performance for a version where complex numbers had been hand-coded with SSE3 assembler instructions, but the C99 variant compiled with Intel icc 10.0 is almost as fast. On the Power5, gcc 3.3.2 gave somewhat better performance for the plain C code, but did not compile the C99 version properly. Evidently,

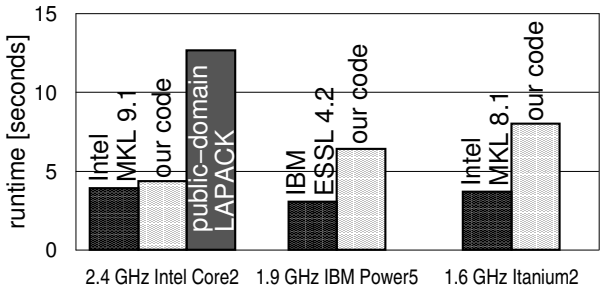

Figure 1. Comparison of runtimes for the serial computation of eigenvalues for a "small" complex matrix in dimension $n = 1184$ between our code and routines `zhpev` provided by different libraries. Note that runtimes include overhead, e.g., for initialization and diagonalization of the resulting tridiagonal matrix.

we are witnessing how complex numbers according to the C99 standard are just being properly implemented in C compilers. C++ is still lagging somewhat behind in performance, but this should also be remedied once the `complex<double>` template defaults to a properly implemented internal data type of the compiler. Overall, we hope that on each platform a C99 compiler will be available soon which yields optimal performance and supports OpenMP such that we will be able to focus on the C99 variant for future code developments. For further analysis in this paper we use the combinations marked by a star in Table 1.

Figure 1 presents a comparison of the serial performance of our code with other libraries. In all cases, the corresponding routine is called `zhpev`, although the interface of IBM ESSL differs from Intel Math Kernel Library (MKL)/LAPACK[9]. It is gratifying to see that we do not only outperform public-domain libraries (like a version of LAPACK included with a recent distribution of Mandrake Linux), but that we can also compete with the commercial Math Kernel Library on the Intel Core2. On the high-performance machines, our code is about a factor 2 slower than the commercial libraries. We can only speculate if this could be improved with better compilers (compare Table 1), but we believe that even serial performance is acceptable at the moment.

Now we move on to discuss parallel performance of our code. For this purpose we use a "large" complex Hermitian matrix of dimension $n = 41\,835$ which is also available from Ref. 23. This matrix requires about 13 GByte of main memory in packed storage mode. On the one hand, this problem is substantially bigger than the problem sizes investigated in other eigensolver performance tests[12,13]. On the other hand, this is still at the lower edge of dimensions where it becomes necessary to use a parallel eigensolver for our purposes[a]. On an IBM p575 with 8 CPUs, our code needs about 14.2 hours real time for the computation of all eigenvalues whereof 99.8% are spent on the Householder reduction[b]. This machine is the one with the Power5 CPUs on which we have previously tested single-CPU performance. So, we can can use n^3 scaling of the runtime measured on our small problem. It turns out that the diagonalization of the large problem takes less than 50% longer

[a] CPU time is too precious to carry out systematic testing for bigger production-type problems[2,3].

[b] We also tested IBM parallel ESSL 3.3[15] under the same conditions. The best performance, namely 9.3 hours real time, was obtained with the routine `pzheevx` running in SMP mode whereas variants with MPI communication were slower than our solution.

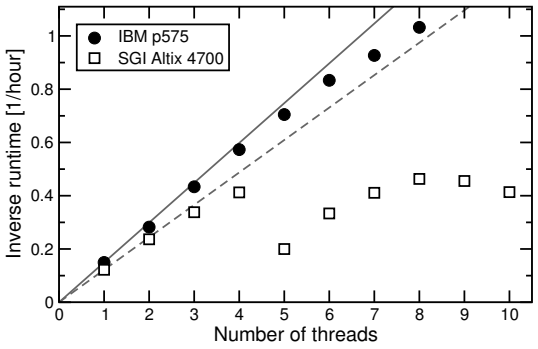

Figure 2. Inverse real runtimes on an IBM p575 and an SGI Altix 4700 for the first 1000 iterations of the Householder transformation for a "large" complex matrix in dimension $n = 41\ 835$. Lines indicate inverse runtimes corresponding to perfect scaling of the single-thread case.

than this optimistic estimate, demonstrating good parallelization and scaling with problem size. More details can be seen in Fig. 2 which shows the inverse real runtime for the first 1000 steps of the Householder reduction as a function of the number of threads. On the IBM p575 we indeed observe good scaling with the number of threads. Fig. 2 also contains results for an SGI Altix 4700. The nodes of this machine consist of 4 Itanium2 CPU Cores whose single-thread performance we have discussed previously. Accordingly, here we observe reasonable scaling for up to 4 threads while the drop in performance between 4 and 5 threads can be attributed to the onset of memory access across the network.

To summarize this section, we have shown that serial performance of our code is competitive, that it scales well to big problems, and that good parallel speedups can be obtained.

4 Discussion and Conclusions

We have presented a parallel implementation of the Householder algorithm for packed complex matrices with proper load balancing and CPU-cache optimization using OpenMP[22]. Our implementation is also able to write checkpoints of the computation onto hard disc which can be used to successively reduce main memory requirements even further in later stages of the computation. A preliminary version of this code without checkpoints has been used[2] to compute the full eigenvalue spectrum of double precision complex matrices in dimension $n = 81\ 752$. With checkpoints it has been possible to push this further[3] to dimension $n = 121\ 968$, and very recently in one case to $n = 162\ 243$. The latter diagonalization required 197 GByte of main memory (mainly for packed storage of the complex matrix) and close to 400 GByte hard-disc space for a fail-save checkpoint. This computation was executed in parallel on a node with 32 1.3 GHz Power4 CPUs where we have measured average CPU efficiencies $\gtrsim 90\%$.

The Householder reduction yields a tridiagonal matrix which can be transformed to real form using simple phase factors. Thus, a diagonalization procedure for real symmetric tridiagonal matrices is needed to finish the computation. Currently, we simply call the LAPACK[9] routine `dsterf` for a reliable computation of all eigenvalues of the tridiagonal matrix. This routine follows the traditional approach provided by the QL/QR-

algorithms[6,10,21]. If only eigenvalues are desired, the diagonalization of the tridiagonal matrix requires a negligible amount of CPU time as compared to the Householder reduction such that optimization of performance is unnecessary. If eigenvectors are also needed, basis transformations have to be computed in the QL/QR-algorithm. Inspection of the QL/QR algorithm shows that this can also be parallelized after putting the rotations into a buffer[c]. Indeed, we already have an OpenMP-parallelized implementation of the QL-transformation for the tridiagonal problem. Recent optimization efforts by other groups have focussed on this diagonalization step of the symmetric tridiagonal matrix[17–20]. Runtime measurements of our not yet optimized QL-transformation show that it requires less CPU time than the Householder step. Faster algorithms for the tridiagonal problem are therefore unnecessary and may even be detrimental for our applications if they go at the expense of reduced numerical accuracy or increased memory requirements[d].

At the moment, the numerical efficiency of our own implementation of the QL-transformation for the tridiagonal problem is still at the level of the routine `tqli` from Ref. 21. As a next step we need to bring this up to the level of LAPACK[10] and implement checkpoints during the diagonalization of the symmetric tridiagonal matrix. It will also be straightforward to derive real variants from our routines although it is not our priority to optimize performance for the real symmetric case. Finally, everything can be canned into a stand-alone package with general-purpose OpenMP-parallelized diagonalization routines which we plan to release into public domain. This package is also scheduled to be integrated in a future release of the ALPS applications suite for strongly correlated electron systems[26,27]. Furthermore, we hope that our code developments will also be useful in other fields such as quantum chemistry.

Acknowledgements

A.H. acknowledges support by the Deutsche Forschungsgemeinschaft through a Heisenberg fellowship (Project HO 2325/4-1). The biggest computations reported here have been made possible by a CPU-time grant at the HLRN Hannover.

References

1. M. E. Zhitomirsky and A. Honecker, *Magnetocaloric effect in one-dimensional antiferromagnets*, J. Stat. Mech.: Theor. Exp., P07012, (2004).
2. F. Heidrich-Meisner, A. Honecker and T. Vekua, *Frustrated ferromagnetic spin-$\frac{1}{2}$ chain in a magnetic field: the phase diagram and thermodynamic properties*, Phys. Rev. B, **74**, 020403(R), (2006).
3. O. Derzhko, A. Honecker and J. Richter, *Low-temperature thermodynamics for a flat-band ferromagnet: rigorous versus numerical results*, preprint arXiv:cond-mat/0703295 (2007).

[c] Such a buffer also helps to optimize CPU cache performance and exists in the corresponding LAPACK[9] routines.
[d] An additional feature of some of these algorithms is the computation of selected eigenvalues only[20]. However, for our purposes (see, e.g., Refs. 1–3) the most important eigenvectors are usually the lowest ones and we start from sparse matrices. Therefore we use completely different algorithms like the Lanczos method[5,25] which can handle much bigger problems if we are interested only in a few eigenvalues (and -vectors).

4. H. Q. Lin, *Exact diagonalization of quantum-spin models*, Phys. Rev. B, **42**, 6561–6567, (1990).
5. N. Laflorencie and D. Poilblanc, *Simulations of pure and doped low-dimensional spin-1/2 gapped systems*, Lect. Notes Phys., **645**, 227–252, (2004).
6. B. N. Parlett, *The Symmetric Eigenvalue Problem*, (Prentice-Hall, 1980).
7. A. Jennings and J. J. McKeown, *Matrix Computation*, (John Wiley, Chichester, 1992).
8. G. H. Golub and C. F. Van Loan, *Matrix Computations*, (Johns Hopkins UP, 1996).
9. E. Anderson *et al.*, *LAPACK Users' Guide, Third Edition*, (SIAM, Phil., 1999).
10. A. Greenbaum and J. Dongarra, *Experiments with QR/QL methods for the symmetric tridiagonal eigenproblem*, LAPACK Working Note 17 (1989).
11. B. Lang, *Direct solvers for symmetric eigenvalue problems*, in: Modern Methods and Algorithms of Quantum Chemistry, NIC Series, Vol. **3**, (Jülich, 2000).
12. I. Gutheil and R. Zimmermann, *Performance of software for the full symmetric eigenproblem on CRAY T3E and T90 systems*, FZJ-ZAM-IB-2000-07, (2000).
13. A. G. Sunderland and I. J. Bush, *Parallel Eigensolver Performance*, CLRC Daresbury Laboratory (`http://www.cse.clrc.ac.uk/arc/diags.shtml`).
14. L. S. Blackford *et al.*, *ScaLAPACK Users' Guide* (SIAM, Philadelphia, 1997).
15. *Parallel ESSL for AIX V3.3, Parallel ESSL for Linux on POWER V3.3, Guide and Reference*, Sixth edition, (IBM, Boulder, 2006).
16. G. Fann, *Parallel eigensolver for dense real symmetric generalized and standard eigensystem problems*, (`http://www.emsl.pnl.gov/docs/nwchem/`).
17. J. J. M. Cuppen, *A divide and conquer method for the symmetric tridiagonal eigenproblem*, Numerische Mathematik, **36**, 177–195, (1981).
18. F. Tisseur and J. Dongarra, *A parallel divide and conquer algorithm for the symmetric eigenvalue problem on distributed memory architectures*, SIAM J. Sci. Comput., **20**, 2223–2236, (1999).
19. I. S. Dhillon, *A New $O(n^2)$ Algorithm for the Tridiagonal Eigenvalue/Eigenvector Problem*, Ph.D. thesis, University of California, Berkeley (1997).
20. I. S. Dhillon and B. N. Parlett, *Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices*, Linear Alg. & Appl., **387**, 1–28, (2004).
21. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, (Cambridge University Press, 1992).
22. R. Chandra *et al.*, *Parallel Programming in OpenMP*, (Morgan Kaufmann, 2000).
23. `http://www.theorie.physik.uni-goettingen.de/~honecker/householder/`
24. `http://gcc.gnu.org/`
25. J. K. Cullum and R. A. Willoughby, *Lanczos Algorithms for Large Symmetric Eigenvalue Computations*, Vol. I (Birkhäuser, Boston, 1985).
26. A. F. Albuquerque *et al.*, *The ALPS Project Release 1.3: Open Source Software for Strongly Correlated Systems*, J. Magn. Magn. Mater., **310**, 1187–1193, (2007).
27. `http://alps.comp-phys.org/`